# CS415 Programming Assignment 4

Adam Lewis

## 1 Overview

In this assignment, you will implement a multi-process, multi-threaded client-server application that allows users to chat among themselves on a local machine or across a network. The applications will use network sockets as a transport mechanism. And then... you get to add a nice little surprise!

## 2 Background

Classical operating systems such as Unix were shared between multiple users. This capability is preserved in modern versions of Unix such as Linux, FreeBSD, and Max OS X. In the days of the mainframe computer, one machine might have had lots of users connected to the machine at any one time.

One of the earliest client-server applications was a chat server. Such systems had a central server process to which clients connected and registered to receive messages. Consider a server that has a chat on-going among users $A$, $B$, and $C$. If user $A$ types a message and hits return, the client process sends this message to all currently connected clients. If a new user $D$ joins the chat, then server will post a message to all connected clients announcing that event. Similarly, if a client leaves the chat, then server announces that fact to the remaining connected clients.

The minimum set of commands that clients and servers must support is (a) connect/login, (b) say something to all participants, and (c) disconnect. Other possible commands are (a) list participants, (b) "whisper" to a single user, (c) an admin command to "kick" a user off the chat, and (d) many more. These commands are optional but both servers and clients need to ignore commands they don't understand and do so without leaking information.

### 2.1 Our problem

We wish to implement our own simple version of this chat system. To begin, we will implement the application as a "in-process" client-server application. Note that we will not concern ourselves at this point with issues of security.

You will write two programs:

**client** A client program will start a new session, listen for messages from the server, display any received messages, and accept simple commands for connection and exiting a conversation.

**server** the server application will listen for requests from clients, process these requests when it receives them from the client, and return data back to the client over the connection. The server must be able to handle multiple connections at the same time.

## 2.2 The design of the server component

Clients and server need to communicate with each other using network sockets. The client and server will need to have some form of locking in place so that a client and the server do not attempt to access the named pipe at the same time.

The server needs to listen for connections over the primary socket, deal with the session level issues for each chat session, and provide some form of logging capability so that a system administrator can monitor what is happening on the client. This means that server must be a multi-threaded application. A thread must be created to listen for new sessions on the main socket and a new thread should be created for each conversation. Conversation threads should be started on the connection, deal with the mechanics of communication, and then go away cleanly when the conversation has completed.

## 2.3 The design of the client component

The client application should be kept as simple as possible. You need only to provide a command-line interface for this application. The client will need to connect to the server process, send and receive messages from the server while connected, and then cleanly disconnect from the server. The client cannot be tied to a single server; this means that a person should be able to connect to a server, hold a conversation with people on that chat server, and disconnect and then reconnect to a different server.

You should implement the following commands:

| Command | Required | Example |
|---|---|---|
| CONNECT | YES | CONNECT username@hostname |
| MESSAGE | YES | MESSAGE Yo, all... what's up? |
| USERS | YES | USERS |
| DISCONNECT | YES | DISCONNECT |
| SEND | OPT | SEND usr1 Yo, dude! That test was hard! |
| KICK | OPT | KICK usr1 |

## 3 Problem statement

Implement this simplified version of chat system using your language of choice. Note that you have to implement both the chat client and chat server.

Oh... I mentioned a "nice little surprise". Add one more command to your chat client: "`PWN`". Sending a command such as "`PWN ls /root`" will result in the server process executing that command on the server and returning the results of the command back to the client over the socket. You may insert evil laugh here.

You are required to implement a sufficient test suite to adequately test your solution. You must submit this test suite (and supporting code) as part of your submission. This also includes providing the sufficient makefiles and supporting scripts to build your program from source.

## 4 Submitting your program

Your submission document must be in PDF format; submission of documents in any other format will result in deduction of points from your grade. Attach your submission to the assignment entry in Blackboard.